

Outline

- **Last time**
 - CPU Scheduling (chapt 6)
 - Assignment #3 out (due Thurs Oct 11)
- **This time**
 - Finish CPU Scheduling (chapt 6)
 - Win scheduling and Assignment #3
- **Next time**
 - IPC Intro: UNIX Signals
 - Synchronization (chapt 7)
- **Midterm: Tues Oct 16 (closed books, closed notes)**

CS414: Operating Systems

Before we start

- **Blake's office hours are on Wed for the next two weeks**
- **Devlab013 has an unexpired Nero on it**
- http://www.devx.com/go-parallel/Article/35471?trk=DXRSS_LATEST (Sept 25)
 - James Reinders is Intel's Chief Evangelist for Intel's Software Development Products. In a recent interview on Devx.com he stated: "If I could get ONE wish fulfilled – it would be for OS scheduling to focus on processes, and not threads, for scheduling. And demand that processes manage their scheduling of threads ... There is a lot of opportunity for operating systems to offer these types of control in the 'running of applications' interfaces. I'd like an OS to let me specify the 'world' my application runs in (which processors, how many, etc.) These interfaces are available in Windows at run time (the task manager will let you adjust where a running task can go)."

CS414: Operating Systems

Recap from last time

- **Scheduling criteria: response time, turnaround time (completion time), "fairness"**
- **algorithm evaluation: deterministic (gant charts), queuing models, simulation, implementation**
- **Non-preemptive policies: FIFO, SJF, Random**
- **Today: preemptive scheduling**
 - Round Robin, Priority, MLQ, MLFQ

CS414: Operating Systems

Preemptive Scheduling Policies

- **First example: Round Robin**
 - The ready queue is treated like a circular queue
 - After each time slice (*quantum*), move the current thread to the back of the queue
 - Selecting a quantum size:
 - what if too big?
 - what if too small?
 - **balance the two!**
- **[Advantage]**
 - each job gets an equal shot at the CPU; shorter jobs finish sooner
- **[Disadvantage]**
 - higher context switching overhead

CS414: Operating Systems

Comparing FIFO and Round Robin

• Assume: time slice 1 second, context switch time 0

Job	Length	Turnaround (FIFO)	Turnaround (Round Robin)
1	100		
2	100		
3	100		
4	100		
5	100		

Case 1

Job	Length	Turnaround (FIFO)	Turnaround (Round Robin)
1	50		
2	40		
3	30		
4	20		
5	10		

Case 2

CS414: Operating Systems

Priority Scheduling

- **A priority is assigned to each job**
- **Always execute the job with the highest priority**
- **Equal priority jobs: FCFS or perhaps RR**
- **In some systems, 1 is the highest priority; in other systems, 1 may be the lowest priority**
- **Usually preemptive; may be nonpreemptive**
- **Problem:**
 - starvation
- **Solution:**
 - aging

CS414: Operating Systems

Multi-Level Queues (MLQ)

- Without feedback, one option is to divide Ready Q into:
 - foreground (interactive, RR) and background (batch, FCFS)
- Assign job priorities that can vary as a f(time)
- Multiple queues with different priorities
- Use RR at each scheduling level; run all jobs in higher queue before jobs in a lower queue
- Longer quantum at lower-priority levels
- Give high priorities to I/O-bound jobs; give low priorities to CPU-bound jobs
- Problem: can still lead to starvation

CS414: Operating Systems

Multi-Level Feedback Queues (MLFQs)

- Feedback: jobs can move between Qs
- Adjusting priorities (note: variations exist)
 - job starts in highest priority Q
 - if quantum expires, drop priority one level
 - if quantum does not expire, move up one level
- Adaptive scheduling policy, because:
 - relies on past behavior;
 - changes in behavior result in changes to scheduling decisions
- Problem: can still lead to starvation

CS414: Operating Systems

Windows Scheduling Principles

- 32 priority levels
- Threads within same priority are scheduled following the Round-Robin policy
- Non-Realtime Priorities are adjusted dynamically
 - Priority elevation as response to certain I/O and dispatch events
 - Quantum stretching to optimize responsiveness
- Realtime priorities (i.e.; > 15) are assigned statically to threads

Thread Scheduling Priorities vs. Interrupt Request Levels (IRQLs)

The diagram illustrates the mapping between Thread Scheduling Priorities (0-31) and Interrupt Request Levels (IRQLs) (0-31). Thread priorities 0-15 are mapped to IRQLs 0-15, which are categorized as software interrupts. Thread priorities 16-31 are mapped to IRQLs 16-31, which are categorized as hardware interrupts. The IRQLs are ordered from highest (31) to lowest (0). The hardware interrupt levels include High, Power fail, Interprocessor Interrupt, Clock, Device n, and Device 1. The software interrupt levels include Dispatch/DPC, APC, and Passive Level.

Priority Adjustments

- Dynamic priority adjustments (boost and decay) are applied to threads in "dynamic" classes
 - Threads with base priorities 1-15 (technically, 1 through 14)
 - Disable if desired with SetThreadPriorityBoost or SetProcessPriorityBoost
- Five types:
 - I/O completion
 - Wait completion on events or semaphores
 - When threads in the foreground process complete a wait
 - When GUI threads wake up for windows input
 - For CPU starvation avoidance
- No automatic adjustments in "real-time" class (16 or above)
 - "Real time" here really means "system won't change the relative priorities of your real-time threads"
 - Hence, scheduling is predictable with respect to other "real-time" threads (but not for absolute latency)

Scheduling Scenarios Quantum Details

- Quantum internally stored as "3 * number of clock ticks"
 - Default quantum is 6 on Professional, 36 on Server
- Thread->Quantum field is decremented by 3 on every clock tick
- Process and thread objects have a Quantum field
 - Process quantum is simply used to initialize thread quantum for all threads in the process
- Quantum decremented by 1 when you come out of a wait
 - So that threads that get boosted after I/O completion won't keep running and never experiencing quantum end
 - Prevents I/O bound threads from getting unfair preference over CPU bound threads

CPU Starvation Avoidance

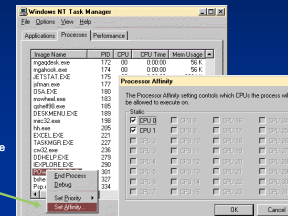
- Balance Set Manager system thread looks for "starved" threads
 - This is a thread, running at priority 16
 - Wakes up once per second and examines Ready queues
 - Looks for threads that have been Ready for 300 clock ticks (approximate 4 seconds on a 10ms clock)
 - Attempts to resolve "priority inversions" (high priority thread (12 in diagram) waits on something locked by a lower thread (4), which can't run because of a middle priority CPU-bound thread (7)).
 - Priority is boosted to 15 (14 prior to NT 4 SP3)
 - Quantum is doubled on Win2000/XP and set to 4 on 2003
 - At quantum end, returns to previous priority (no gradual decay) and normal quantum
 - Scans up to 16 Ready threads per priority level each pass
 - Boosts up to 10 Ready threads per pass
 - Like all priority boosts, does not apply in the real-time range (priority 16 and above)



Hard Affinity

- Affinity is a bit mask where each bit corresponds to a CPU number
 - Hard Affinity specifies where a thread is permitted to run
 - Defaults to all CPUs
 - Thread affinity mask must be subset of process affinity mask, which in turn must be a subset of the active processor mask

- Functions to change:
 - `SetThreadAffinityMask`,
 - `SetProcessAffinityMask`,
 - `SetInformationJobObject`
- Tools to change:
 - Task Manager or Process Explorer
 - Right click on process and choose "Set Affinity"
 - Psexec -a



Soft Processor Affinity

- Every thread has an "ideal processor"
 - System selects ideal processor for first thread in process (round robin across CPUs)
 - Next thread gets next CPU relative to the process seed
 - Can override with:


```
SetThreadIdealProcessor (
    HANDLE hThread, // handle to the thread to be changed
    DWORD dwIdealProcessor); // processor number
```
 - Hard affinity changes update ideal processor settings
 - Used in selecting where a thread runs next (see next slides)
 - For Hyperthreaded systems, new Windows API in Server 2003 to allow apps to optimize
 - `GetLogicalProcessorInformation`
 - For NUMA systems, new Windows APIs to allow applications to optimize:
 - Use `GetProcessAffinityMask` to get list of processors
 - Then `GetNumaProcessorNode` to get node # for each CPU
 - Or call `GetNumaHighestNodeNumber` and then `GetNumaNodeProcessorMask` to get the processor #s for each node

PA#3: Scheduler

- If you're re-using your code base, make sure you get rid of any debugging mods from PA#1 and PA#2 (
 - Just start again from scratch if you want.
- Fair-share scheduling can be approximated by random-process-then-random-thread
 - This design contains many trade-offs: which ones? You decide.

CS414: Operating Systems

Our approach

- Take action whenever a thread stops executing on the CPU:
 - Re-insert the just-ending thread (as normal) into its position in the MLFQ
 - Manually select the realtime thread that you want to execute next, if one exists (via "random process then random thread")
 - Manually put the selected realtime thread at the front of the appropriate queue

CS414: Operating Systems

Manually Selecting the RT Thread

- Find all of the realtime threads in the MLFQ of WRK.
- Divide the threads up according to the processes to which they belong.
- Print out which # process and then which # thread you're going to manually select.
- Put this thread in now at the beginning of the queue.

CS414: Operating Systems

